Università della Svizzera italiana

Institute of Computing
CI

High-Performance Computing Lab **Institute of Computing**

Student: Marenco Turi Gualtiero Discussed with: Sobolev Mark, Bella Jonatan, Perpili Suuraj

Solution for Project 3

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

This project will introduce you a parallel space solution of a nonlinear PDE using OpenMP.

1. Task: Implementing the linear algebra functions and the stencil operators [35 Points]

1.1. Implementation's notes

The main notable thing here is that I used the already made function for the dot product, since the $L2$ norm can also be written as $X^t X$ for a vector X, which becomes the sum of the square of the entries of the vector. The other implementations are rather straightforward and can be consulted in the code.

1.2. Results

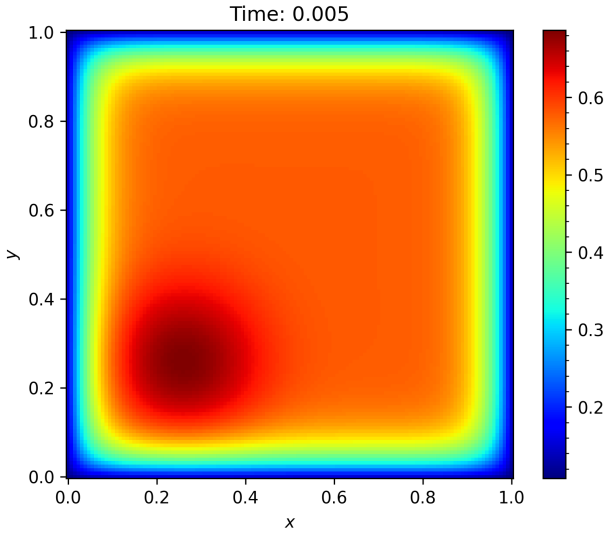
The test has been run using the requested parameters:

- One node
- Exclusive mode
- 100 time steps
- Simulation time 0.005 s
- 128 Domain size

The results are the following:

```
                    Welcome to mini-stencil!
version   :: C++ Serial
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.223262 seconds
1511 conjugate gradient iterations, at rate of 6767.82 iters/second
300 newton iterations
--------------------------------------------------------------------------------
### 1, 128, 100, 1511, 300, 0.223262 ###
Goodbye!
```

The plot of the population concentration at time 0.005 returned by the python code is the following:



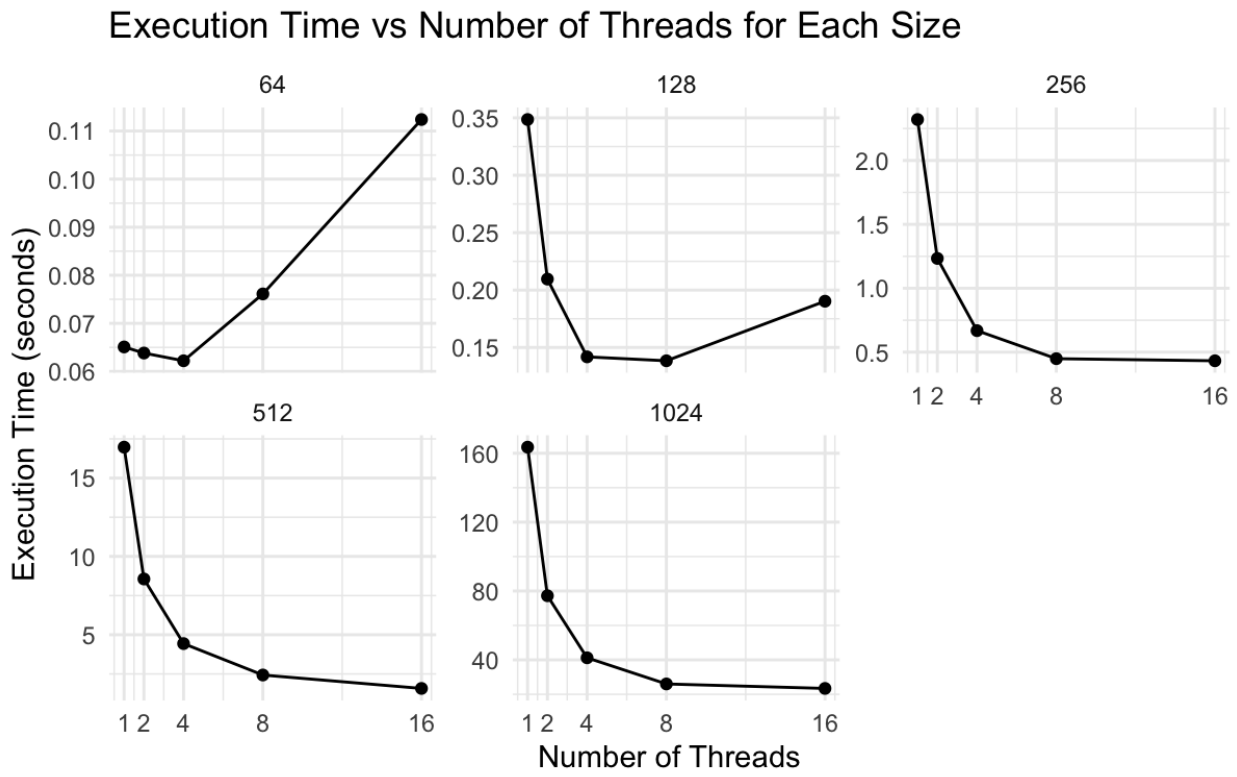
Which, as required, is mostly identical to the given one.

2. Task: Adding OpenMP to the nonlinear PDE mini-app [50 Points]

2.1. Implementing Parallelization

If the program id running OpenMP, the number of thread will be printed, otherwise the mini-app will inform that everything is working in serial mode. Most of the implementations of OpenMP are pretty straightforward and require just the *#pragma* statement and sharing all the variables. The main thing to be noted is that for the dot product function, on the opposite than everything else, there is race condition risk. Using reduction on the result variable to share the it in proper order between the threads solve the issue. For the operator file, everything is again immediate. It would be difficult to consistently get bit-wise identical results, as it would require control over the order in which the values are combined, which is not available without compromising the efficiency of parallelization. With a unspecified order for joining the values, rounding errors may in fact deviate the result, however the difference is small and can be ignored.

2.2. Strong Scaling



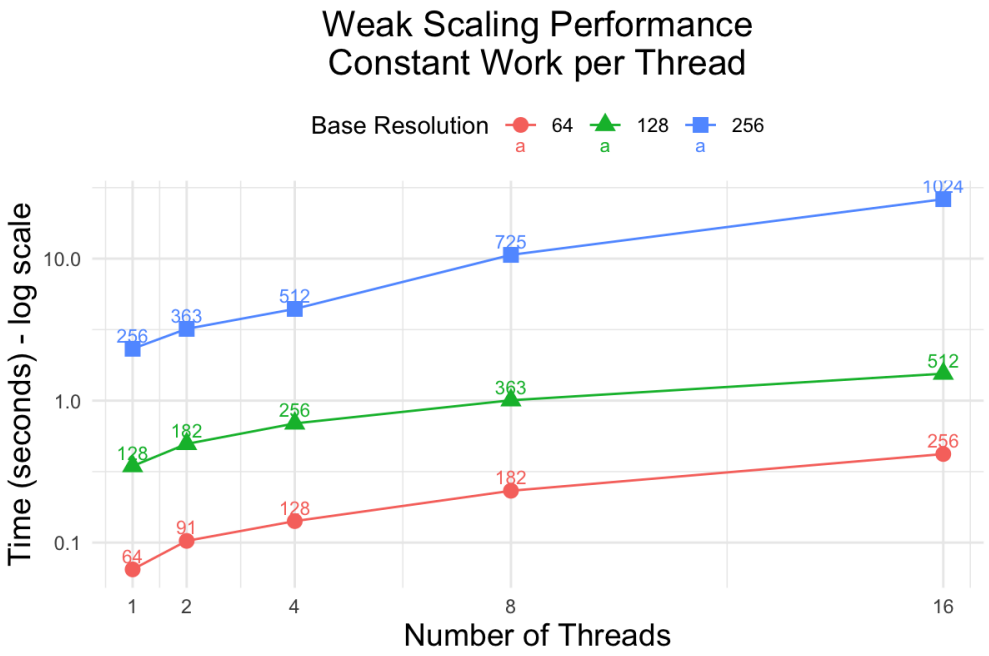
Analyzing the data a clear pattern emerges: the efficiency gains from parallel processing become more pronounced as the dataset size increases, which is even more evident for bigger number of threads. This is due to the relationship between workload size and the parallelization overhead. In smaller datasets, the time saved through parallel execution can be minimal or even offset entirely by the overhead associated with managing multiple threads. This mens that in some cases, particularly with smaller problem sizes and for bigger numbers of threads, parallelization can introduce delays rather than accelerate computation.

For instance, with the 64x64 size dataset, the cost of coordinating multiple threads adds a fixed overhead to the execution time. In these smaller problem sizes, the parallel processing workload per thread is limited, resulting in less opportunity for individual threads to contribute substantial speedup. This may means that parallel execution may be not only inefficient but also increase execution time, as seen in the case of the 64x64 size, where the parallelization yield a negative impact on performance for more than 4 threads.

However, as the problem size grows, these overhead costs are increasingly compensated for by the larger amount of work, making parallelization more effective and the gains more substantial. Consequently, larger datasets demonstrate an increasingly favorable response to parallel execution in general but to bigger number of threads in particular, effectively utilizing the multi-threaded approach to achieve significant speedups.

2.3. Weak Scaling

Here the aim is to analyze scaling with constant work per each thread.



To analyze scaling with a constant work-per-thread approach, the problem sizes are set up so that as the number of threads N_{CPU} increases, the total problem size expands proportionally. This method ensures that each thread consistently handles the same amount of work, regardless of the thread count. Here's how it looks for each base resolution:

Base Resolution 64x64

- With 1 thread: starts at 64×64.
- With 2 threads: problem size is 91×91.
- With 4 threads: problem size is 128×128.
- With 8 threads: problem size is 181×181.
- With 16 threads: problem size is 256×256.

Workload per thread: approximately 4,100

Base Resolution 128x128

- With 1 thread: starts at 128 x 128 .
- With 2 threads: problem size is 181 x 181 .
- With 4 threads: problem size is 256 x 256 .
- With 8 threads: problem size is 362 x 362 .
- With 16 threads: problem size is 512 x 512 .

Workload per thread: approximately 16400

Base Resolution 256x256

- With 1 thread: starts at 256 x 256 .
- With 2 threads: problem size is 362 x 362
- With 4 threads: problem size is 512 x 512
- With 8 threads: problem size is 724 x 724 .
- With 16 threads: problem size is 1024 x 1024 .

Workload per thread: approximately 65600.

In an ideal scaling scenario, it would would be expected that doubling the number of threads when doubling the problem size accordingly would result in a constant time to solution. This is because each additional thread would theoretically absorb the extra work associated with the larger grid, maintaining the workload per thread and balancing the computational effort across threads.

However performance is limited by several overhead factors associated with parallelization. Each additional thread introduces some overhead costs due to the need for coordination and communication among threads.